Melodee: Solving ODEs with platform-specific code generation

DOE COE Performance Portability Meeting

Robert Blake

August 24, 2017

LLNL-PRES-737110

Lawrence Livermore National Laboratory

Melodee is...

Modular **E**xpression **L**anguage for **O**rdinary **D**ifferential **E**quation **E**diting

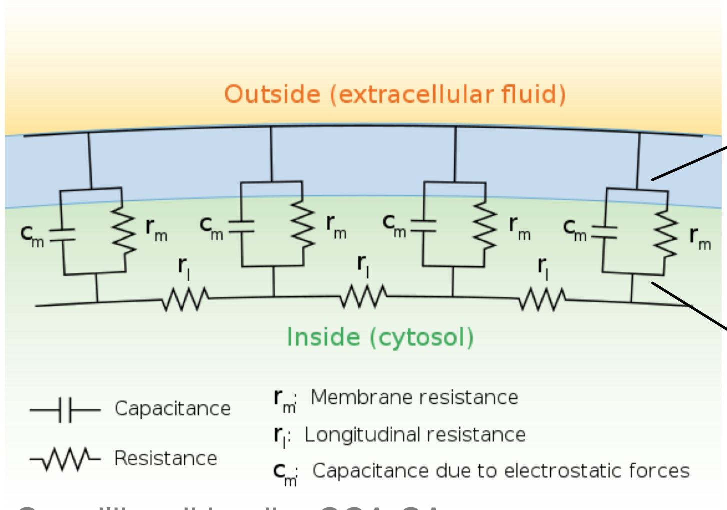
For users	For developers
A language for describing ordinary differential equations	A code-generation toolkit for ordinary differential equations

Overview

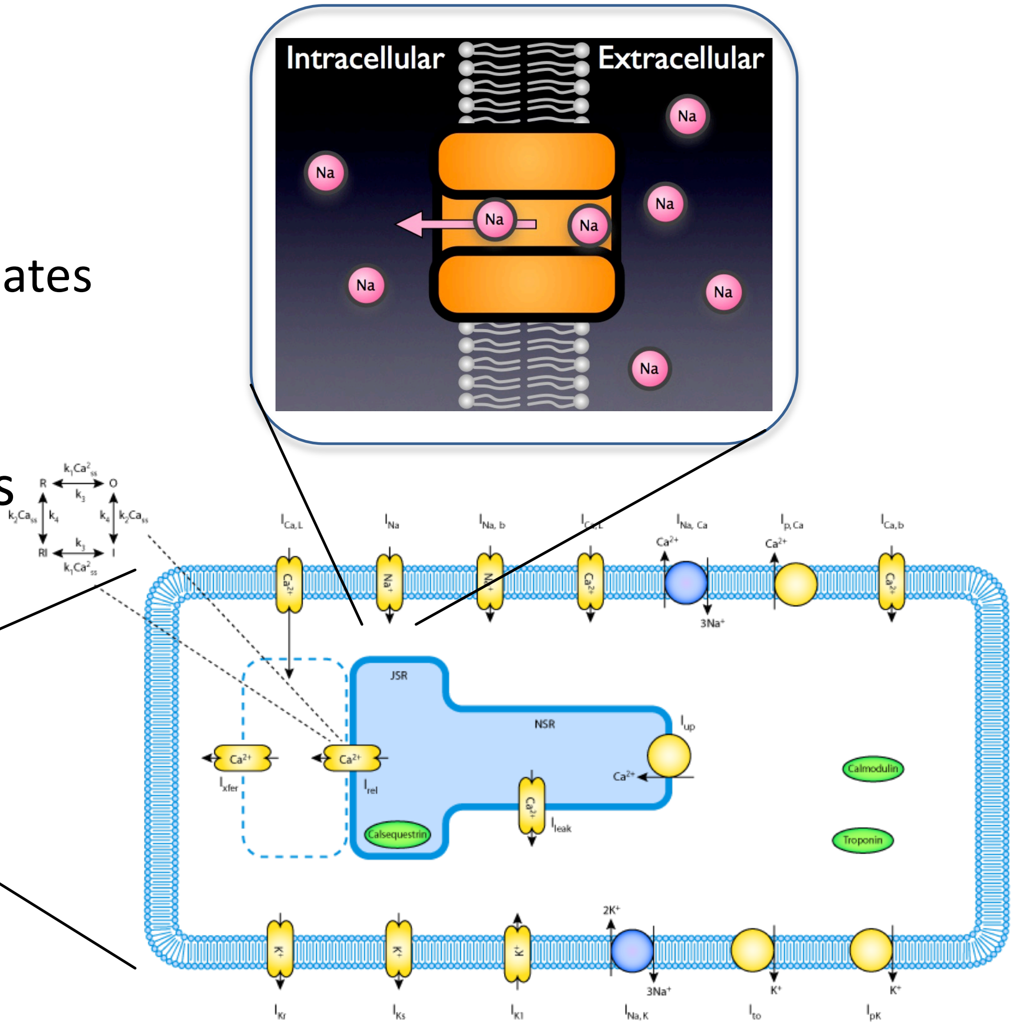
- Motivating problem– cardiac electrophysiology
- Design of the language
- Using Melodee to generate platform specific code
  - GPUs
  - CPUs
  - KNL
- Advantages of DSLs and code generation

Reaction term for cardiac electrophysiology

- Embarrassingly parallel ODEs
- Computation-bound
- Each cell requires
  - ~ 20-60 differential variable updates
  - ~ 60-100 libm evaluations
  - ~ 150-500 equation calculations
- Reaction takes 80% of the flops


Outside (extracellular fluid)
c_m
r_m
r_l
Inside (cytosol)
Capacitance
Resistance
r_m: Membrane resistance
r_l: Longitudinal resistance
c_m: Capacitance due to electrostatic forces

Smedlib, wikipedia, CCA-SA



It’s hard to port platform-specific code

- Cardioid was a Gordon Bell Finalist
- Exactly one reaction model optimized for BGQ
  - Math functions replaced with hard-coded rational polynomials
  - Thread load balancing based on BGQ architecture
  - Lots of BGQ vector intrinsics
  - 5800 LOC for 173 equations
- Our job: port this to GPUs

Our portability woes will only get worse

- The reaction model changes constantly
  - Reaction models are under constant development
  - Every experiment needs a novel reaction model
- We need platform-specific optimizations for performance
- ...but optimized code is
  - Un-maintainable
  - Platform dependent
  - Man-hour devouring
  - Tedious to write

Melodee: a language for ODEs

- Melodee is a domain specific language for describing ordinary differential equations
- Not Turing complete
- http://github.com/llnl/melodee

```
subsystem lorenz {
  sigma = 10;
  beta = 8/3;
  rho = 28;

  diffvar x,y,z;
  x.init = 1;
  y.init = 1;
  z.init = 1;

  x.diff = sigma*(y-x);
  y.diff = x*(rho-z)-y;
  z.diff = x*y-beta*z;
}
```

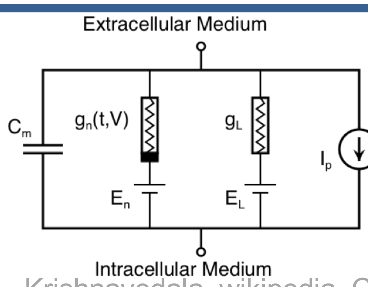
Design goals for Melodee

- No existing language fit our needs, so we made our own

cellML BioNetGen

- Goals
  - **Separate** math from implementation
  - **Compatible** – easy to convert from C, Matlab, and cellML
  - **Agnostic** – independent of domain & simulator
  - **Extendible** – separate domain knowledge from semantics
  - **Modular** – encourage re-use for large ODEs
  - **Safe** – unit checking for common mistakes
- Would this be useful for your domain?

Hodgkin Huxley in Melodee



Krishnavedala, wikipedia, CC0

```
integrate time {ms};

shared V {mV};
shared Iion {uA/uF};
shared E_R {mV};

subsystem hodgkin_huxley {
  subsystem leakage_current {
    E_L = E_R+10.613;
    @param g_L = 0.3;
    provides accum Iion += g_L*(V-E_L);
  }
  subsystem potassium_channel {
    diffvar @gate n {1};
    alpha_n = -0.01*(V+65)/expm1(-(V+65)/10);
    beta_n = 0.125*exp((V+75)/80);
    n.init = 0.325;
    n.diff = (alpha_n*(1-n)-beta_n*n);
    E_K = (E_R-12);
    @param g_K = 36;
    provides accum Iion += g_K*n^4*(V-E_K);
  }
  subsystem sodium_channel {
    diffvar @gate h {1};
    alpha_h {1/ms} = 0.07*exp(-(V+75)/20);
    beta_h {1/ms} = 1/(exp(-(V+45)/10)+1);
    h.init = 0.6;
    h.diff = (alpha_h*(1-h)-beta_h*h);
    diffvar @gate m {1};
    alpha_m = -0.1*(V+50)/(exp(-(V+50)/10)-1);
    beta_m = 4*exp(-(V+75)/18);
    m.init = 0.05;
    m.diff = (alpha_m*(1-m)-beta_m*m);
    E_Na = (E_R+115);
    @param g_Na = 120;
    provides accum Iion += g_Na*m^3*h*(V-E_Na);
  }
}
subsystem membrane {
  provides diffvar @interp(-100,100,2e-2) V;
  provides V_init {mV} = -75;
  Cm = 1;
  i_Stim = 0;
  if (time >= 10 && time <= 10.5) {
    i_Stim = 20;
  }
  V.init = V_init;
  V.diff = -Iion+i_Stim;
}
subsystem full_model {
  use hodgkin_huxley {
    export E_R as V_init;
  }
  use membrane;
}
```

Melodee is a code generation toolkit

- Developer writes code generators in python
  - Cardioid generator is only 600 LoC!
- Melodee parses .mel models for you
  - gives you a list of expressions in SSA
- Sympy for symbolic manipulation
  - Free symbolic differentiation.
- Flexible
  - Backwards compatible with cellML
  - Used in 3 different simulators

Reaction model optimizations

- **Rational polynomials** – replace expensive function evaluations with faster functions
- **Kernel fission vs fusion** – separate the ODE into multiple functions or one function
- **Replace exp/log** – variants based on floating point binary representation
- **Intrinsics** –use the compiler to vectorize or do it ourselves
- **SoA vs AoS** – How do we lay out our data structures?

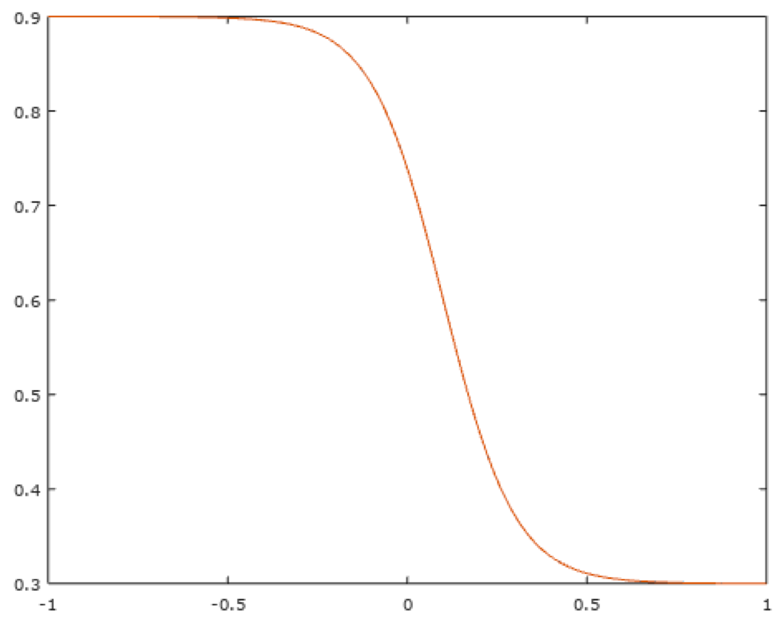
Optimization	BGQ	P100	KNL
Rational polynomials	yes	yes	no
Kernel fission vs fusion	fission	fusion	fusion
Replace exp/log	yes	no	no
Explicit vectorization with intrinsics	yes	no	yes
SoA vs AoS	SoA	SoA	AoS

Rational polynomials can replace expensive functions

```
double Afcaf = 0.3+0.6/(1.0+exp((v-10.0)/10.0));
```

becomes

```
double Afcaf;
{
   double numerCoeff[]={-9.52275328672 ... };
   double denomCoeff[]={2.18001528726e ... };
   double numerator=_numerCoeff[0];
   for (int jj=1; jj<8; jj++)
      _numerator = numerCoeff[jj] + v*numerator;
   double _denominator=denomCoeff[0];
   for (int jj=1; jj<6; jj++)
      _denominator = _denomCoeff[jj] + v*denominator;
   Afcaf = numerator/denominator;
}
```



GPU: Embedding the coefficients is much faster

```
poly(double *in,
     int np, double *p,
     double *out)
{
  int ii = blockIdx.x*blockDim.x + threadIdx.x;
```

np=60
in[1e6]
out[1e6]

Memcpy:
30.940us

```
out[ii]=in[ii];
```

Naïve:
202.15us

```
double z = 0;
for (int k=np-1; k>=0; k--)
   z = p[k] + z*in[ii];
out[ii] = z;
```

```
/* 0x2b8 */
{ IADD32I R3, R3, -0x1;
  LDG.E.64 R10, [R6]; }
  ISETP.GT.AND P0, PT, R3, RZ, PT;
  IADD32I R6.CC, R6, -0x8;
  IADD32I.X R7, R7, -0x1;
  DFMA R4, R8, R4, R10;
 @P0 BRA 0x2b8;
```

Embedded:
40.760us

```
double *my_p[] = {...};
double z = 0;
for (int k=np-1; k>=0; k--)
   z = my_p[k] + z*in[ii];
out[ii] = z;
```

```
DFMA R8,R2,R8,c[0x2][0x68];
DFMA R8,R2.reuse,R8,c[0x2][0x60];
DFMA R8,R2.reuse,R8,c[0x2][0x58];
...
```

Unrolling with a duff's device

Unrolled

```
__constant__ double c_p[];
...
double z = 0;
switch (np) {
   case 8: c_p[7] + z*in[ii];
   case 7: c_p[6] + z*in[ii];
   case 6: c_p[5] + z*in[ii];
   case 5: c_p[4] + z*in[ii];
   case 4: c_p[3] + z*in[ii];
   case 3: c_p[2] + z*in[ii];
   case 2: c_p[1] + z*in[ii];
   case 1: c_p[0] + z*in[ii];
   default:
}
out[ii] = z;
```

- On CPUs, this is
  - just as fast as embedding
  - uses run-time coefficients
- On GPUs
  - c_p must be constant memory
  - c_p must be a constexpr
  - ptxas doesn't emit indirect branches
    - Still have to pay for performance
    - Memcpy: 30us
    - Embedded: 40us
    - Unrolled: 46us

Embedded coefficients are faster and simpler on GPUs

Polynomial code on CPUs

- Ran 1M points for 2k iterations for 15 degree polynomial
  - All times in seconds (lower is better)
- Vec means using explicit vector intrinsics

Description	gcc	icc	clang
embedded	6.81	16.56	6.94
naïve	21.94	23.95	22.27
unrolled	14.46	14.67	14.239
embedded+vec	6.79	6.86	6.87
naïve+vec	7.35	7.91	10.12
unrolled+vec	6.83	6.82	6.86

Explicit vectorization must be used for performance on CPUs

Rational polynomial summary

- BGQ, Haswell: need manual vectorization
- GPU: coefficients must be known at compile time
- KNL: rational polynomials are slower?
  - I see a 10% slowdown currently
  - KNL has vector intrinsics for exp, etc.
  - Rational polynomials cause L1 spills??

Intrinsics

- BGQ, Haswell, KNL
  - Compilers will NOT auto-vectorize this code
  - Must generate vector intrinsics specific to platform
- GPU
  - No intrinsics necessary

Kernel fission or fusion?

- BGQ: required separating the reaction model into decoupled functions
  - Each thread integrated different variables independently

- KNL, GPUs: Faster results by putting everything in one monolithic kernel
  - Hide memory latency with computation.

Replacing exp/log

- Use fact that IEEE754 contains a base-2 logarithm in exponent

- BGQ: essential for good performance
- GPU: replacing exp/log is no faster.
- KNL: 50% slowdown when replacing exp/log
  - Faster intrinsics on chip

Data layout

Structure of Arrays

```
struct state {
  double x[n];
  double y[n];
}
```

Array of Structures of Vectors

```
struct stateVec {
  double x[vwidth];
  double y[vwidth];
}
stateVec state[n/vwidth];
```

- Haswell, KNL: AoSoV is faster
- BGQ, GPU: SoA is faster

Summary

- Domain specific languages make us more productive
- Code generation is essential for portable performance

Optimization	BGQ	P100	KNL
Rational polynomials	yes	yes	no
Kernel fission vs fusion	fission	fusion	fusion
Replace exp/log	yes	no	no
Explicit vectorization with intrinsics	yes	no	yes
SoA vs AoS	SoA	SoA	AoS

	BGQ	P100	KNL
Performance (% of peak)	60%	38%	11%

http://github.com/llnl/melodee

Acknowledgements

- LLNL
  - Dave Richards
  - Tom O’Hara
  - Xiaohua Zhang
- IBM
  - Doru Bercea
- Intel
  - Doug Jacobsen

Lawrence Livermore National Laboratory

Use case: Developing a new reaction model

```
integrate time {ms};
shared V {mV};
shared Iion {uA/uF};
shared V_init {mV};

subsystem ORd_with_newIKr {
  shared ko {mM};
  shared EK {mV};
  shared IKr {uA/uF};
  use ORd - .sodium_current - .rapid_rectifier_current {
    export ENa;
  }
  use TT06.fast_sodium_current {
    export i_Na as INa;
    export E_Na as ENa;
  }
  use newIKr;
}
```